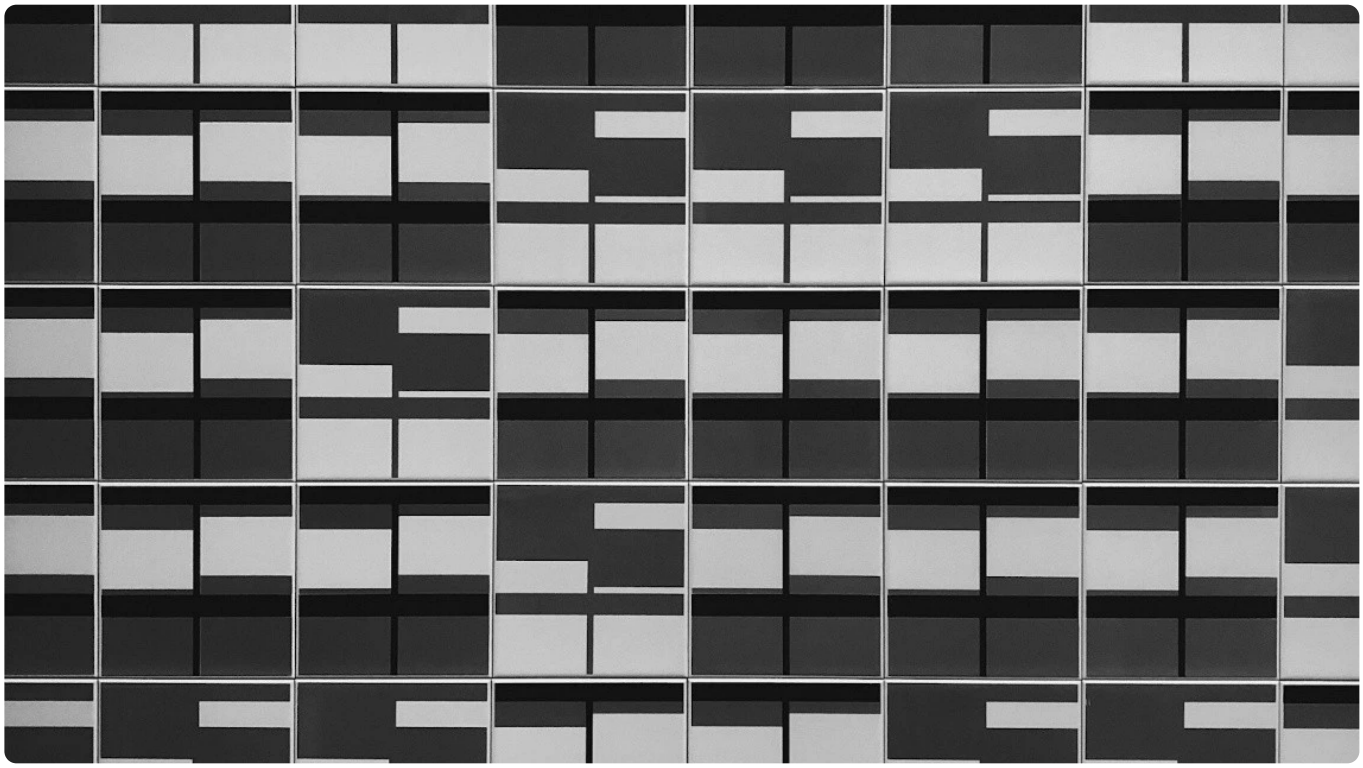
29 | Dataflow（一）：正确性、容错和时间窗口

2021-12-15 徐文浩

《大数据经典论文解读》 课程介绍 >



讲述：王惠

时长 21:27 大小 19.66M

你好，我是徐文浩。

在 Storm 的论文里，我们看到 Storm 巧妙地利用了异或操作，能够追踪消息是否在整个 Topology 中被处理完了，做到了“至少一次（At Least Once）”的消息处理机制。然后，在 Kafka 的论文里，我们又看到了，Kafka 通过将消息处理进度的偏移量记录在 ZooKeeper 中的方法，使得整个消息队列非常容易重放。Kafka 的消息重放机制和 Storm 组合，就使得 At Least Once 的消息处理机制不再是纸上谈兵。

然而，我们并不会满足于“至少一次”的消息处理机制，而是希望能够做到“**正好一次（Exactly Once）**”的消息处理机制。因为只有“正好一次”的消息处理机制，才能使得我们计算出来的数据结果是真正正确的。而一旦需要真的实现“正好一次”的消息处理机

制，系统的“**容错能力**”就会变得非常重要。Storm 的容错能力虽然比起 S4 已经有了一定的进步，但是实际上仍然非常薄弱。

所有的这些问题，伴随着 Kappa 架构设想的出现，为我们带来了新一代的流式数据处理系统。那么，接下来的几节课里，让我们步入现代流式数据处理系统，一起看看从 Google 的 MillWheel、Dataflow，到开源的 Apache Flink 的系统是怎么回事儿。

在这节课里，我们会先看看在没有这些系统的时候，在实践上使用 Storm 时会遇到哪些实际的问题。其实也正是由于这些问题，催生了现代流式处理系统的诞生。在学完这节课之后，我希望你能够理解以下这三点：

为什么“Exactly Once”的消息处理是困难的，但又是必须的。

为什么 Storm 的容错机制，比起实际需求远远不够，和 MapReduce 这样同样粗糙而原始的框架比起来，Storm 的容错机制缺失了什么。

“时间窗口（Time Window）”是一个什么样的概念，为什么这个概念对于流式数据处理系统如此重要。

在理解了这些问题之后，我们其实就已经开始逼近现代的流式数据处理系统了。只要能够解答好这些问题，我们就会有一个全新的系统了，而这样一个全新的系统究竟应该如何搭建，其实就是后续我们会详细讲解的 MillWheel、Dataflow 以及 Flink 的核心知识点了。

一个简单的流式数据处理系统

我们先来看一看，在有了 Kafka 和 Storm 之后，一个实际的流式数据处理系统是怎么样的。

最简单的，我们就采用一个进行广告点击率计算和计费的数据处理需求。我们的日志会是这样的格式：

广告位ID	广告客户ID	广告ID	事件类型	用户UID	事件ID	时间戳	花费（分）
12345	1A2B3C	ABCDE	Impression	U231231	E1234567	1600000123	0
67890	5E6F7G	MNOPQ	Click	U789789	E1234568	1600000456	35

极客时间

广告系统日志的简单示例

每一条日志，都表示一次广告相关的日志。其中，广告位 ID+ 广告客户 ID+ 广告 ID 标明了是哪一个广告，展示在了哪一个广告位置上。比如，可口可乐的新年促销广告，展示在了某视频网站的开屏页，就可以通过这三个字段分辨出来。

事件类型这个字段，用来标注这条日志是表示一次广告的展示，还是一次广告的点击。

用户 UID，用来标识是哪一个用户。这个在实践层面，可以方便我们对于同一个用户，在短时间内反复点击相同的广告进行去重。

事件 ID，用来标识一个唯一的事件。在实践层面，如果出现系统故障，我们常常会重试保障“至少一次”的数据处理。而有了这个字段之后，我们就可以在处理的时候进行去重，这样我们才有可能做到“正好一次”的数据处理。

最后的时间戳字段，用来记录事件发生的时间。花费字段，则是记录这一次点击广告，需要花费广告客户多少预算。

这里的日志，只是一个最简化的模型。在实际的广告系统中，会有上百个字段，比如我们还会记录 IP 地址，以分辨用户所在的地理位置等等。不过，有了这个最简单的日志格式，我们已经可以做两个最常见的广告数据的流式处理了。

第一个，自然是接近于实时的广告计费了。广告客户会设置当天可以花掉的广告预算，我们不能让它花费超过设置的预算，所以我们必须实时地统计客户的花费。

第二个，则是统计各个广告的点击率，对于点击率太低的广告，我们应该反馈给广告投放系统，自动停止广告的投放。

那么根据这两个需求，我们就可以很容易地基于 Kafka 和 Storm，搭建起一个我们需要的流式数据处理系统。

首先，前端的应用服务器，会把产生的广告日志发送给一个负载均衡。然后通过负载均衡，均匀而随机地发送给 Kafka 不同的 Broker 服务器。下游有一个 Storm 集群，里面有一个 Topology，同时完成了广告计费，以及广告的点击率统计的工作。

这个 Topology，就只有简单的两层。

第一层是一个 **KafkaSpout**，它会从 Kafka 拉取日志，然后解析并获取需要的字段，并向下游的 Bolt 进行数据分发。

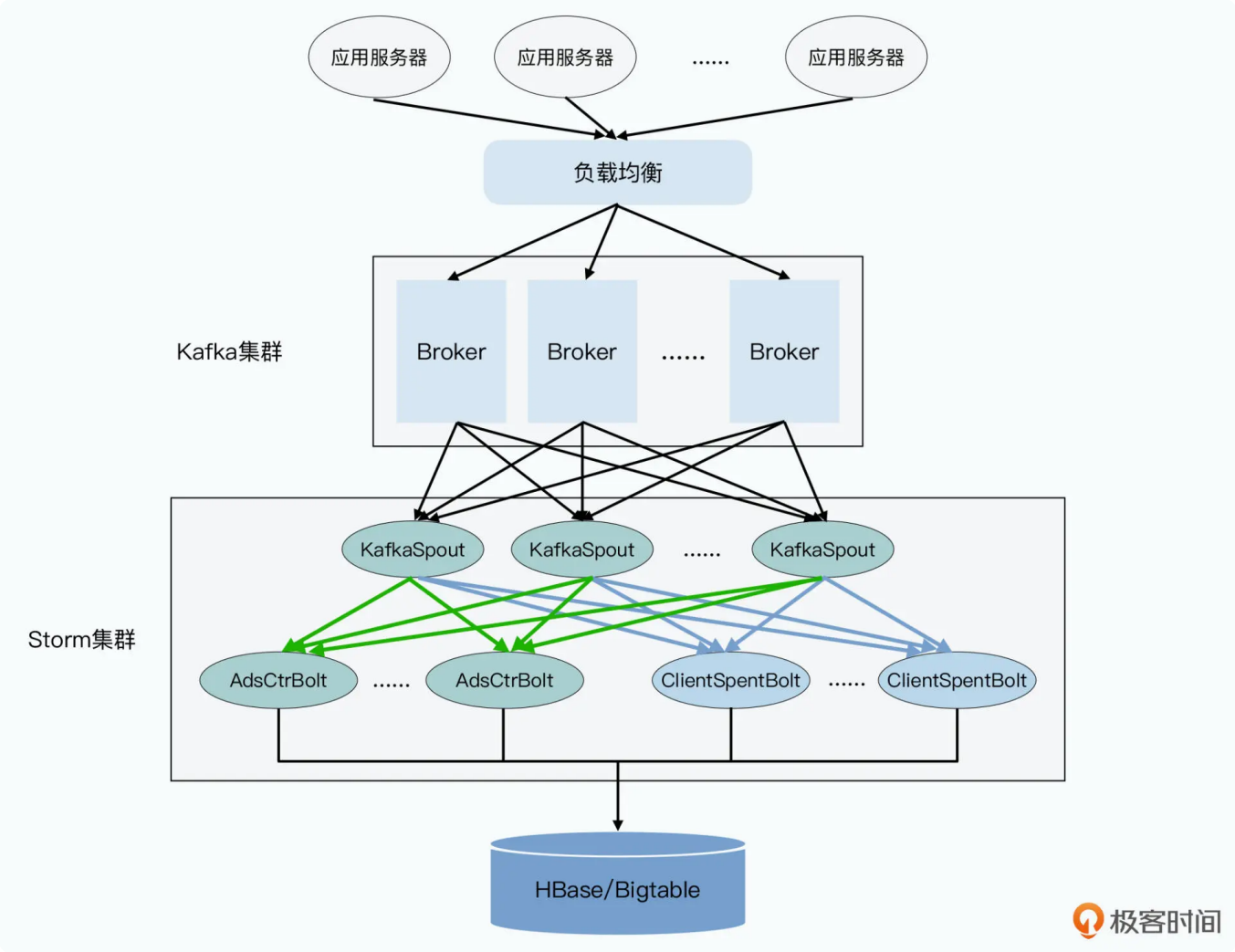
KafkaSpout 的每一条日志，都会发送两条消息给下游两种不同的 Bolt。一条发给 **AdsCtrBolt，用来统计不同广告的点击率**；另一条发给 **ClientSpentBolt，用来计算每个广告客户的花费**。

在向下游发送数据的时候，都是采用**字段分组**的方式。发给 AdsCtrBolt 的，是按照广告 ID 进行分组，发给 ClientSpentBolt 的，则是按照广告客户 ID 进行分组。这样，所有相同广告的日志，都会发送到同一个 AdsCtrBolt 里；而所有相同广告客户的日志，也都会发送给同一个 ClientSpentBolt。

AdsCtrBolt 的处理逻辑很简单，就是它会在内存里维护一个 `广告ID=>(展示次数，点击次数，广告花费）` 的 Map。然后定时把这个表输出到外部的数据库里，比如 HBase 或者 Bigtable 这样的数据库。也就是，它会每分钟输出一次对应广告 ID 的点击率信息。

ClientSpentBolt 的逻辑也很简单，就是它可以以更高的频率，比如每秒，甚至每次接收到一次广告点击，就对应更新一次 HBase 里的广告花费数据。

最后，整个 Storm 的 Topology，是开启了 AckerBolt 的，也就是我们会确保所有的消息能够至少被处理一次。



“正好一次”的正确性

这样，一切看起来都很完美，我们简单地通过 Kafka+Storm，就有了一条可以实时计算广告花费和广告点击率的数据流水线。当然，如果我们的系统非常稳定，没有任何软硬件故障的时候，事情也许是这样的。不过，**在大数据领域，我们始终面临“出错”这个问题。**而一旦出错，我们的麻烦就来了。

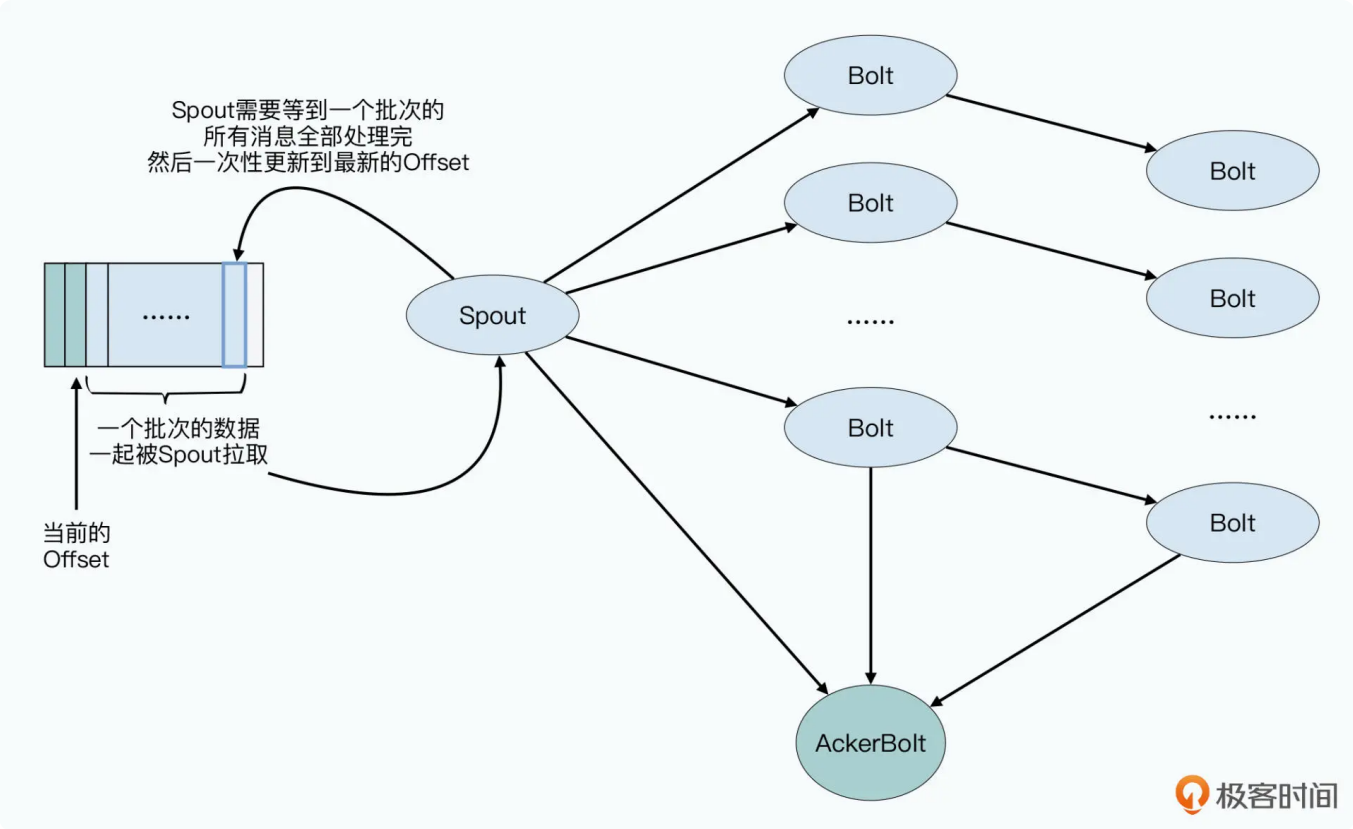
首先就是这个“至少一次”数据处理的特性，其实已经满足不了我们实际的业务需要了。随着时间的推移，我们已经把“广告计费”这样对于准确性要求很高的应用，也放到流式处理系统里来。

在我们这个应用场景里，可能某一个 ClientSpentBlot 写入外部数据库的时候，出现了比较高的延时。这个时候，Storm 的“至少一次”的处理机制，会重发对应的消息。如果没有考虑这样重发的消息，那么我们就会在 ClientSpentBolt 里面，重复计算同一条日志的广告花费，这就意味着我们多扣了广告客户的预算，这显然是难以接受的。

而如果说，单条日志重发计费，可能对于最终计费的影响还很小。那么如果 Storm 的某一个 KafkaSpout 出现了硬件故障，挂掉了，我们就可能有一大批消息会重复计费了。

因为为了性能考虑，我们从 Kafka 拉取数据，不会是拉一条、处理一条，然后更新一次 ZooKeeper 上的偏移量。特别是 ZooKeeper 会受不了这么大的负载，**它和 Chubby 一样，是用于实现一个粗粒度的分布式锁**，而不是一个高性能的 KV 存储。所以，KafkaSpout 会从 Kafka 拉一小批数据，然后发送出去，等到这一小批数据发送完了，并且下游都处理完了，才会变更一次 ZooKeeper 上的偏移量。

但是，只要其中有一条消息在下游还没有处理完的时候，KafkaSpout 所在的服务器挂掉了，对应的偏移量没有更新。那么在容错机制下，重新启动在另一台服务器上的 KafkaSpout，会重新再发送一遍这一批数据。而这个时候，我们就不只是重新对一条日志重复计费，而是需要对一大批日志重复计费。



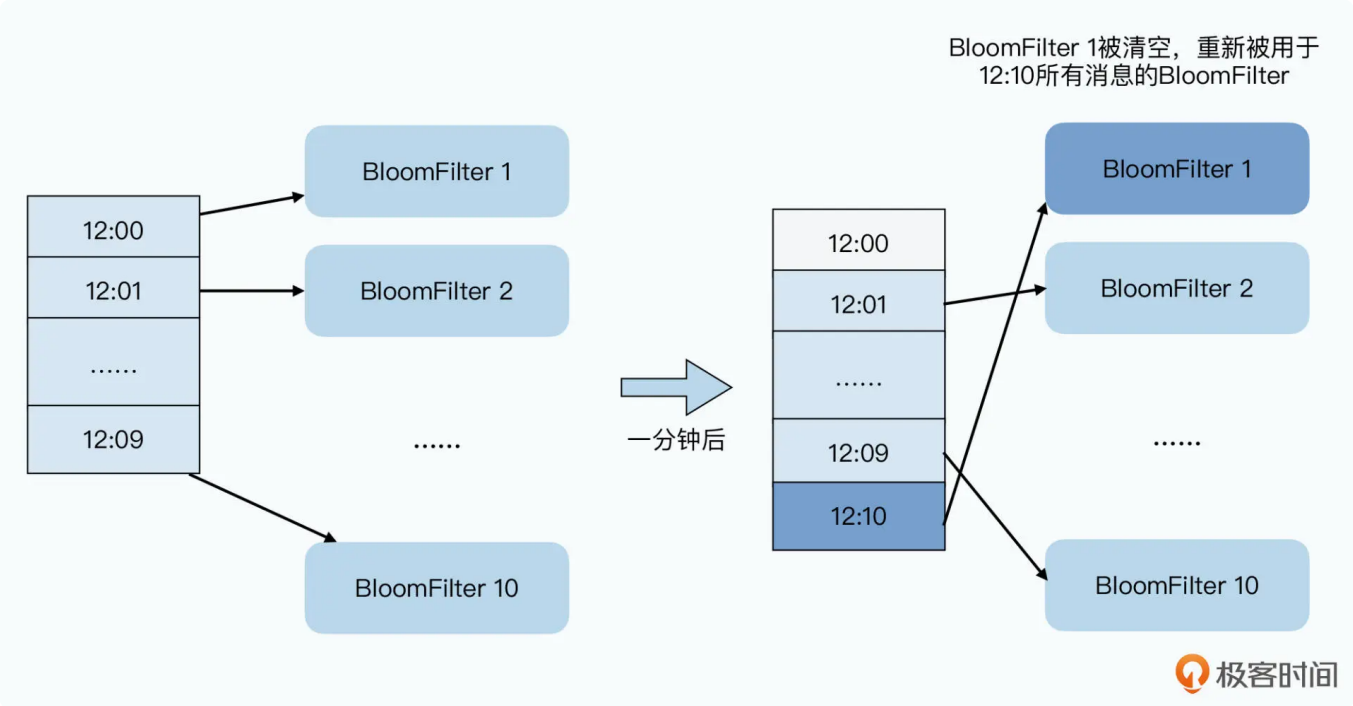
在Kafka+Storm的组合下，我们也不是每条消息都去更新Offset，而是一个Batch一个Batch地更新
在这个过程中如果Spout挂了，那么整个Batch的数据都会重新再发送一遍

要解决这个问题，一个很直观的思路，自然是**对重复发送的日志或者消息进行去重**。最简单的方式，就是在每一个 Bolt 里，我们维护一个这个 Bolt 已经处理完成的，所有的 message-id 的集合。那么，任何一条新的消息发送过来的时候，我们都去这个集合里看一看，这条消息是否已经处理过了，就能解决这个问题了。

不过，让每个 Bolt 都保留所有处理过的 message-id 的集合，显然会**占用太多的内存**了。因为在流式系统里，随着时间的推移，系统处理过的日志量在不断地增加，message-id 的集合只会越来越大。所以，在工程实践上，我们可以做两个优化：

第一个，是**使用 BloomFilter 进行去重，来代替原始的数据集合**。我们把所有已经处理过的 message-id 放到一个 BloomFilter 里去，这样可以大大压缩我们需要的内存空间。不过，使用 BloomFilter 会带来的副作用是，我们可能会有很小的概率误算，使得不是重复的消息也会被认为是重复的。

第二个，是**把数据按照时间窗口，切分成多个 BloomFilter**。比如，我们可以设定有 30 个 BloomFilter，每个 BloomFilter 都只存放某一分钟的 message-id。而每过一分钟，我们都把 30 分钟前的那个 BloomFilter 清空。这样，我们可以通过一个固定大小的内存空间，确保只要是 30 分钟内的重复数据，就不会被多次处理。因为一般来说，简单的重发，不太可能超过 30 分钟。我们可以根据系统的实际情况，来设定这个对应的时间窗口。



通过按照时间范围，切分多个BloomFilter，在时间过期之后，回收BloomFilter供新的时间分片使用

真正做到“正好一次”的数据处理，是现代流式数据处理的第一个目标。

计算节点迁移的容错问题

BloomFilter 的引入，使得我们用于计算的 Bolt 节点，其实有了“状态”。也就是说，它自身已经不是一个纯粹的函数了。事实上，不仅是为了做到“正好一次”的消息处理需要

状态，我们本身的数据处理需求就需要状态。

比如，我们的 AdsCtrBolt 里，维护的那张 广告ID=>(展示次数，点击次数，广告花费) 的 Map，也就是我们在 Bolt 里维护的状态。不过，需要维护状态又给我们带来了一个新的挑战，那就是系统的**容错问题**。

对于系统的“计算节点”的容错很容易，我们只要在另外一台服务器上，重新启动一个 Bolt 就好了。但是这个时候，我们之前维护在 Bolt 内存里的 广告ID=>(展示次数，点击次数，广告花费) 的状态就已经丢失了。如果我们是每一分钟输出一次数据给 HBase/Bigtable 里的话，这意味着我们经常会丢掉一分钟的数据。

事实上，不仅仅是针对容错问题，我们需要考虑恢复 Bolt 里的状态，对于系统的可扩展性，我们同样需要考虑恢复 Bolt 里的状态。Storm 的论文里，我们的并行度是部署 Topology 的时候预先设定好的。但是，这样的系统，很难进行动态的扩容。

如果我们的广告业务越来越红火，意味着上游的日志越来越多。这个时候，我们其实希望调整每一层并行度，通过增加并行度，使得我们系统仍然能够在线水平扩展。

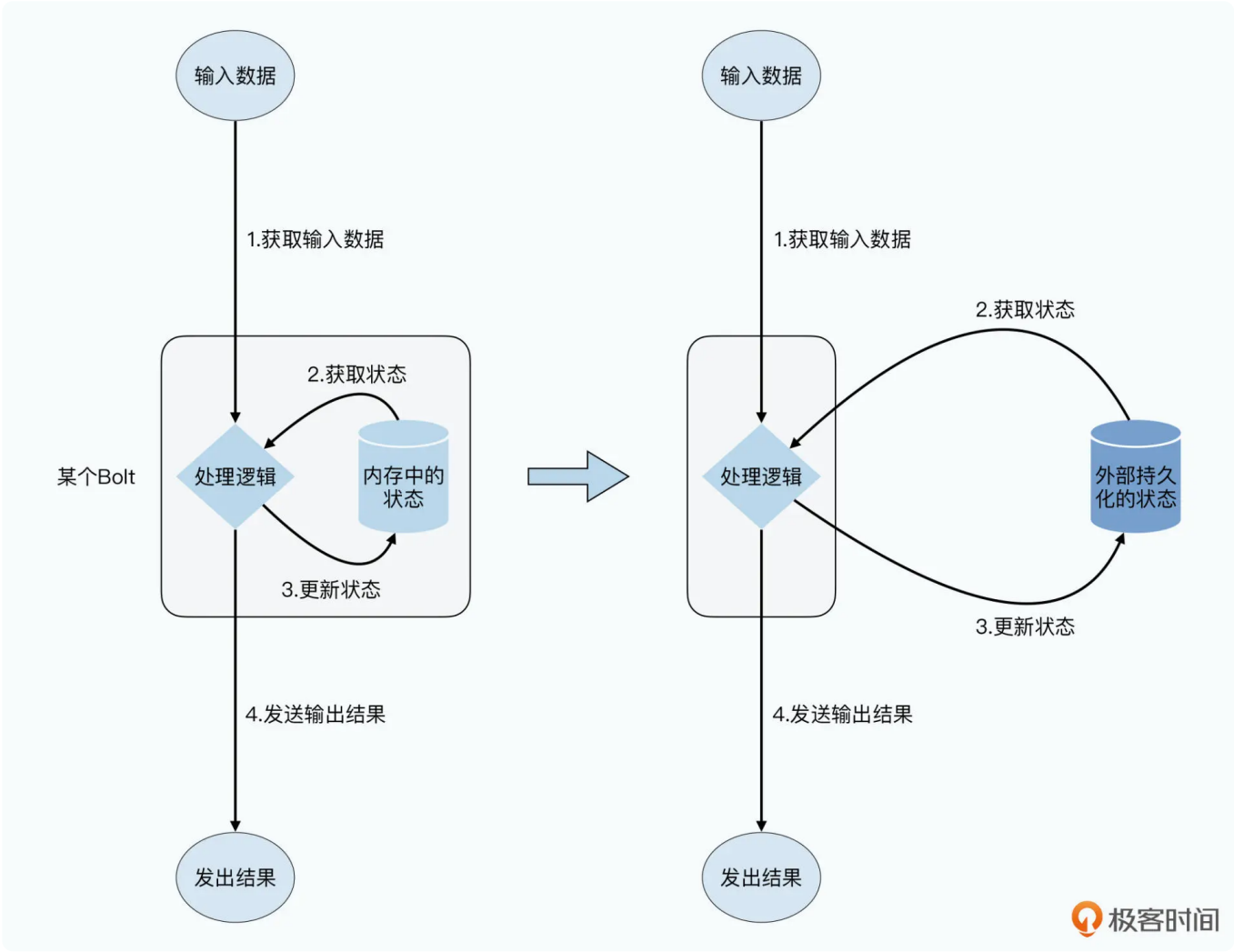
但是，要调整并行度，意味着两点：

第一点，意味着我们会**在线上增加服务器的数量**，有些正在运行的 Bolt 会被迁移（Migrate）到其他的服务器上去。

更进一步地，我们**想要增加 Bolt 的数量**。这意味着，Bolt 里的这个 广告ID=>(展示次数，点击次数，广告花费) 的状态，也需要能够拆分。这个时候，S4 的设计反而显得更合理，那就是每一个 PE 都对应着一个 Key 了。这样，我们需要迁移的状态，就和对应的计算函数是绑定在一起的了。

Bolt 会被拆分和迁移，并且在迁移的过程中，我们需要能够保留状态信息，这意味着我们的状态需要能够**持久化**下来。我们需要能够把这些状态，也更新到一个稳定的外部存储中去。当我们的节点挂掉，在其他服务器上恢复计算能力的时候，需要把这些状态信息重新读取回来。

并且，这个能力，也使得我们去调度计算变得更容易了，我们可以动态地在线上增加系统的并行度。而不是采用部署一个新系统再把老系统下线，这样运维成本更高的模式。



为了确保系统的真正容错，我们需要把Bolt这样的计算单元的状态信息持久化到外部存储中去

通过把各个计算节点的中间状态持久化，使得系统在容错情况下，仍然能够做到“正好一次”的数据处理，并且能够在线上动态扩容、调度计算，是现代流式数据处理的第二个目标。

处理消息的时间窗口

除了重复发送的消息去重，Bolt 的中间状态需要持久化之外，其实我们前面的 Topology 还有一个问题没有解决好，这个问题就是“时间问题”。

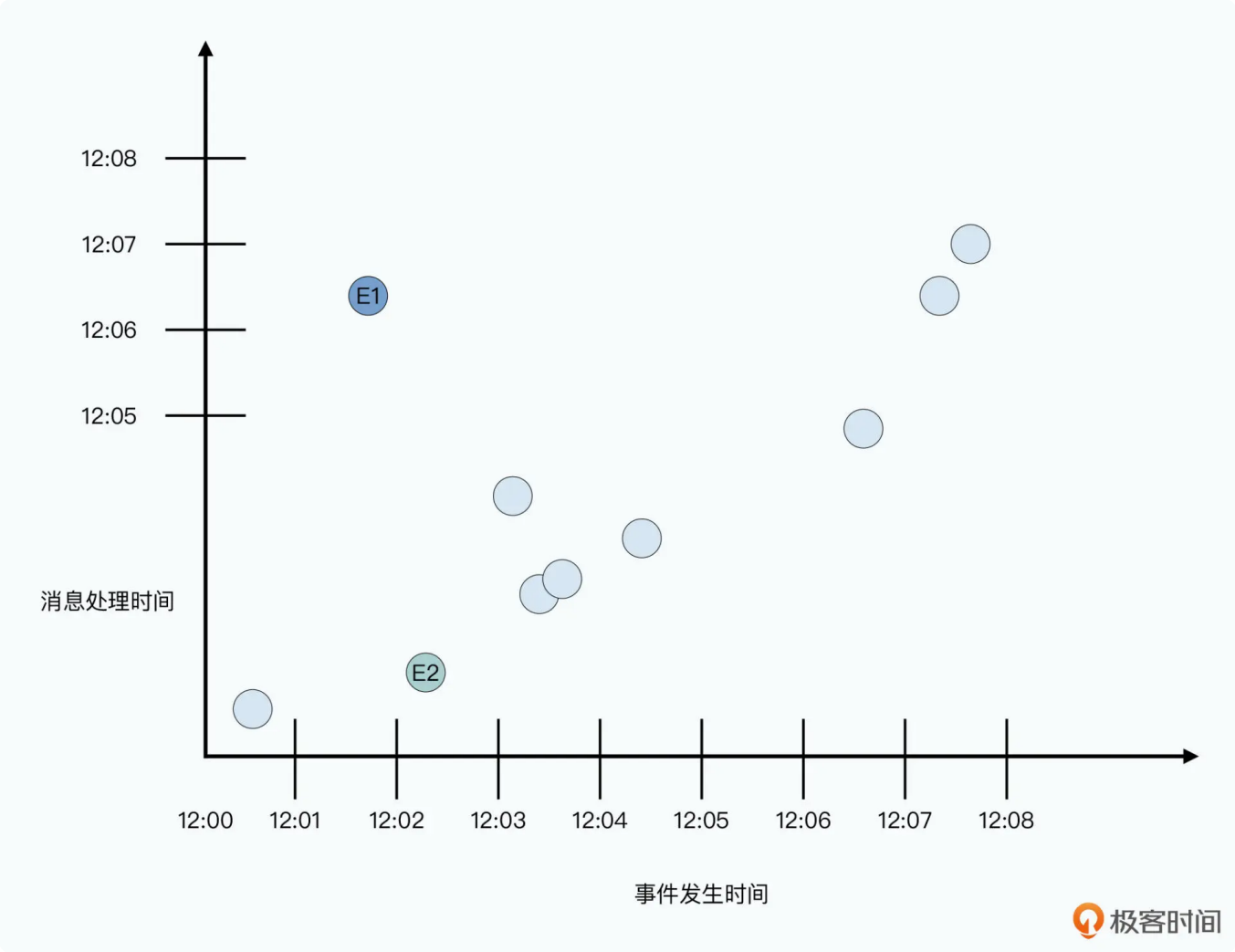
我们在前面 Storm 的 Topology 里，很简单地用一句“每分钟”输出一次广告点击率，概括了 AdsCtrBolt 的逻辑。这个“每分钟”的时间，依靠的是 Storm 内建的一个叫做 **TickTuple** 的机制。Storm 可以在系统层面设置一个时间间隔参数，根据这个参数，Storm 会按照固定的时间间隔，向每一个 Bolt 和 Spout 发送一个特殊的 TickTuple。我们的 Bolt 只需要每当接收到这个 TickTuple 的时候，把当前计算出来的状态信息输出出去就好了。

但是，这个处理逻辑有一个问题，就是我们用消息传输到 AdsCtrBolt 的时间，替代了对应的广告曝光和点击发生的时间。也就是我们用**处理时间**（Processing Time）替代了**事件时间**（Event Time）。这样，我们计算出来的点击率，乃至计费信息，会和实际情况有差异。

而且，这个差异情况，在很多场景下我们是无法容忍的。

一种情况是和业务需求相关，比如我们的广告客户，设置了广告预算都在 11 月份花完。那么，在 11 月 30 日晚上 11 点 59 分 59 秒发生的广告点击，实际被处理的时候很有可能已经是 12 月 1 日了。这样，我们的广告客户会看到，他并没有在 12 月份分配任何广告预算，但是我们的系统却让他在 12 月 1 日有了花费，这显然会引起客户的不满。

另一种情况，则是和我们对于日志的重放相关。无论是系统故障，还是我们修改了数据分析逻辑，当我们要通过重放 Kafka 里的日志，重新计算统计数据的时候，现在的逻辑会造成更大的麻烦。因为所有的日志都是在短时间内重放，所以我们会把过去几小时，甚至是几天的数据，都统计在最近几分钟内，我们的统计数据不是有一些小小的误差，而是完全错误的。



在实际的流式数据处理中，从事件发生，到消息被处理会有一定的延时
而且消息并不一定会按照它们发生的时间顺序被处理
在该图里，事件E1比E2早发生，但是要晚上好几分钟才被处理

当然，批量重放日志，不是一个常见的情况。但是，在硬件故障的情况下，部分前端应用服务器的日志没有及时进入 Kafka，或者某些 Kafka 的 Broker 的部分日志，没有及时进入我们的 Topology 则是一个常见的情况。在这样的场景下，我们仍然会有大量的日志，出现少则数十秒，多则一两个小时的延误。这样计算出来的错误的统计数据，我们仍然接受不了。

一个合理的解决方案，就是我们需要使用实际的事件发生的时间（即 Event Time），来进行相应的数据统计。但是这样一来，我们就面临两个新的问题。

第一个问题，是我们不能简单地维护 `广告ID=>(展示次数，点击次数，广告花费)` 这样一个映射关系了，而是需要一个 `时间窗口=>[广告ID1=>(展示次数，点击次数，广告花费)，广告ID2=>(展示次数，点击次数，广告花费) ，……]` 这样一个三维多层的映射关系了。

第二个问题，是我们很难决策，什么时候应该将我们的统计结果，写入到外部的数据库里。因为在上节课里我们就看到过，上游发送过来的日志，并不是严格按照时间排序的。一个可行的方案，就要考虑很多因素，比如我们要加上这样几个判断条件和因素：

> 我们需要在 Bolt 内部，维护一个“时钟”，来判断最近接收到的日志，大概是在什么时间戳附近。比如，我们可以用最近 1000 条日志的最早时间戳，作为这个时间戳。当然，这里的 1000 是一个我们自己可以设计的参数。
>
> 当我们的时间窗口，已经比最近的时间戳晚上一个特定的时间长度，比如 5 分钟了。我们可以认为，接下来不会再接收到这样的日志了，那么我们就可以把对应的数据写入到外部的数据库里去。
>
> 而如果又有一条我们确认不再应该收到的日志又传输过来了，那么我们有两个选择。一个是直接忽略丢弃，另一种，则是从我们的数据库里读取出之前的统计信息，更新之后再写回到数据库。

显然，要实现这些逻辑，我们使用 Storm 现有的内置机制是做不到的。

虽然我们还是可以通过像 TickTuple 这样的机制，定时提醒我们去检查是否应该把数据从 Bolt 内存里维护的 Map，输出到外部的数据库里。但是，像是维护时间窗口的映射关系、统计最近日志的时间戳这些逻辑代码，我们仍然都需要自己来撰写。

而我们希望的仍然是，大数据应用的开发人员只需要撰写统计相关的业务逻辑代码，而不需要为了容错，或者考虑 Kafka 发送数据可能存在的延时，去写大量实现容错功能的代码。

我们希望能够把和时间窗口相关的，以及触发数据更新到外部数据库相关的处理机制，在流式处理框架中内建。而撰写流式数据处理逻辑的开发人员，不需要关心这些机制和容错问题，这个也就是现代流式数据处理的第三个目标。

小结

好了，相信到这里，你对于流式数据处理面临的挑战应该已经清楚了。可以看到，面对这些挑战，我们原本以为已经非常优秀的 Storm 是远远不够的。

我们需要一个系统，能够达成三点目标：

第一点，是**“正好一次”的数据处理机制**，要做到这一点，我们需要在流式数据系统里，内置一个数据去重的机制。

第二点，是**把计算节点需要使用的“状态”信息持久化下来**。这样，我们才能够做到真正的容错，而不是在系统出错的时候丢失一部分信息。而且，这个机制也有助于我们在线扩容。

第三点，是我们需要**把流式数据处理的时间窗口，以及触发机制内置到流式处理系统里面去**。这样，我们就可以让我们的业务代码，专注于实现业务逻辑，而不是需要自己在应用代码里，搞一套时间窗口的维护和触发机制。

在这节课里，我们已经给出了一些实践上的解决方案。但是，我们并不希望，自己在写 Storm 的 Spout 代码的时候，写上一大堆代码，来解决正好一次的数据处理、Spout 中间状态的持久化，以及针对时间窗口的处理逻辑。因为这些问题，是流式数据处理的共性问题。

我们希望能有一个流式处理系统，帮助我们解决这些问题。作为应用开发人员，我们仍然只需要撰写业务代码。这个，也就是我们接下来会讲解的 MillWheel、Dataflow 以及 Flink 的系统会做到的事情。

推荐阅读

那么，聊到现代流式数据处理，我们绕不开《Streaming System》这本书，目前国内也已经出版了影印版。当然，整本书的篇幅很长，我推荐你可以先去读一下作者写的两篇小短文 Streaming 101和 Streaming 102这两篇文章。

思考题

在“正好一次”的数据处理中，我们引入了 BloomFilter，来减少对于内存的占用。但是，BloomFilter 始终还是会存在误算的情况，这个可能会导致我们少计算对应的数据。那么，你能设计一组参数，估算一下 BloomFilter 可能会导致我们漏掉计算多少日志吗？

欢迎在留言区分享你的答案和思考，也欢迎你把今天的内容分享给更多的朋友。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 28 | Kafka（二）：从Lambda到Kappa，流批一体计算的起源

下一篇 30 | Dataflow（二）：MillWheel，一个早期实现

精选留言 (3)

写留言

那时刻

2021-12-15

估算一下 BloomFilter 可能会导致我们漏掉计算多少日志吗。我觉得需要日志总数据 以及BloomFilter的 false positive的概率。而计算false positive概率需要 BloomFilter 中bit的总数，储存元素的个数，hash函数的个数

展开

1

在路上

2021-12-16

徐老师好，Wikipedia Bloom filter在Probability of false positives一节，给出了漏算日志率的经典估算公式，p=(1 - (1 - 1/m)^kn)^k，其中n为布隆过滤器要处理的日志条数，m为布隆过滤器的bit位数，k为日志映射到布隆过滤器的hash函数个数。
论文《ON THE FALSE-POSITIVE RATE OF BLOOM FILTERS》(2008年)讨论了经典公式在什么情况下有效，m的值要足够大，同时k的值要足够小。论文《A New Analysis of the F...

展开

小明

2021-12-15

我仿佛看到了 我在360三年的时光走马灯

展开